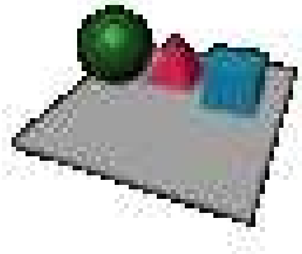
Lecture 11: Thursday, 9 October 1997

Today:
  Recursive Ray Tracing (H&B 14.6, 14.8, 4.8)
  Asst 5 Out (Due 5pm Fri, 17 Oct)
Description on course web page
Next Week:
  Tuesday's lecture: CG techniques, videos
  Tuesday evening: Haptic device demo, 34-101, 7-9pm
  Thursday lecture: Final project brainstorming

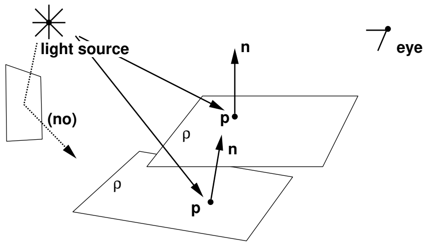
Scan Conversion and Shading

Shading computed with a *constant* amount of state
  Typically, some number of h/w light sources

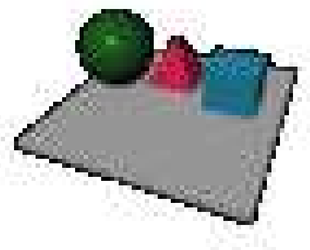


Consequences:
  plastic look; incorrect highlights; no
    shadows
    secondary illumination
    transmission
    focusing effects

Hardware Shading



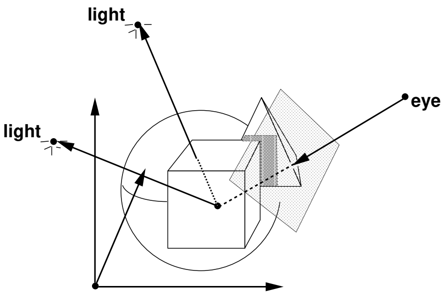
Point light sources only
No occlusion testing
Primary light sources only

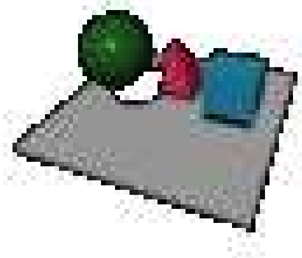


```
% ivray raycast.iv -e raycast.env -s L
```


Alternative: Ray Casting, Semi-Local Shading

Idea (Appel, 1968):
  Cast ray from eye through each pixel
  Determine closest object along ray
  Shade by summing *unoccluded* lights
Non-recursive! (But improved quality, realism)
(Primary) shadows easily handled

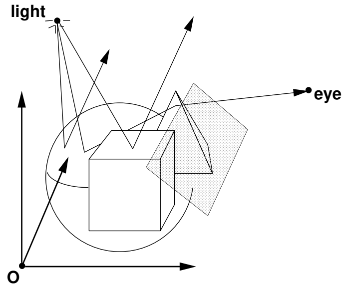




```
% ivray raycast.iv -e raycast.env -s S
```

Recursive Ray Tracing

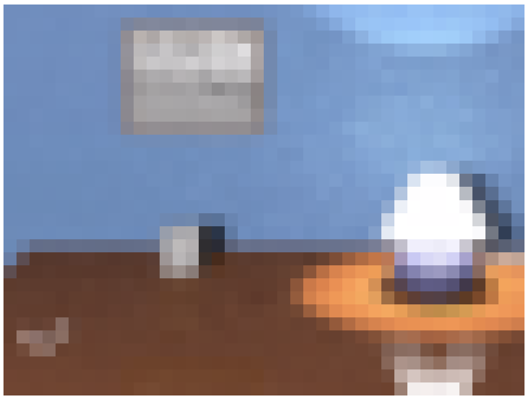
Extend to reflection, refraction
  Clearly shading must take entire scene into account
  Ray tracing is a "global illumination algorithm"
Idea: Light *originates* at light sources, so
  "trace" photon paths from the light source
Known as **Forward Ray Tracing:**
  At each interaction, surface properties dictate
    absorption, reemission, transmission probability



Disadvantages?

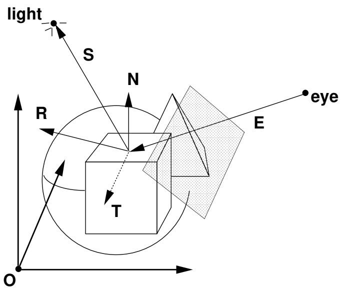
Forward Ray Tracing

Disadvantages
  very few of the photons end up at the eye
  very hard to know in which directions
    photons should be sent
  enormous number of cycles expended per photon
    (can be ameliorated by *packet tracing*)
  result is usually objectionable *noise*



Backward Ray Tracing

Insight: we only "see" rays that make it to eye
So, trace "eye rays" **E** *backward* into scene
  Find contributions to shading at surface points



```
% ivray raycast.iv -e raycast.env -s R -d 1
```

  Shadow rays **S** (to light sources)
  Reflection rays **R** (along specular direction)
  Refraction rays **T** (along refraction direction)
Note: Shading operation is **recursive** !

Backward Ray Tracer Pseudo-Code

```
RayTrace ( frustum, viewport ) {
  For each raster y
    For each pixel x
      E = ray from eye through pixel x, y
      FrameBuffer[x][y] = Trace ( eye, E, 1 )
} // RayTrace
```

Note: `RayCast()` now returns `hit`:
  scene object `hit->object`
  intersection parameter `hit->t`
  surface point `hit->P`, normal `hit->N`

```
Radiance Trace ( Point R, Ray D,
                 int depth ) {
  Hit *hit = RayCast ( R, D );
  if ( hit )
    return Shade ( hit->object, D, hit->P, hit->N, depth )
  else
    return Background ( R, D )
} // Trace
```

Backward Ray Tracer Pseudo-Code

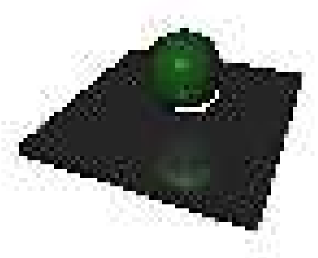
```
// Shade obj surface at point ShadeP, normal ShadeN,
//      as seen along direction Along
Radiance Shade ( Object obj, Ray Along,
        Point ShadeP, Vector ShadeN, int depth ) {
  Radiance r;

  r = ambient radiance;
  For ( each light ) {
    Ray sRay = Ray from point to light;
    if ( sRay . ShadeN > 0 ) {
       r += diffuse contribution * light visibility
       r += specular contribution * light visibility
       }
    }

  // terminate recursion
  if ( depth == maxDepth ) return r;

  // continued
  ...
```


Backward Ray Tracer Pseudo-Code

```
  ...
  // Shade() continued

  if ( object reflective ) {
    Radiance rRefl;
    Ray rRay = reflection ray
    rRefl = Trace ( ShadeP, rRay, depth + 1 )
    r += rRefl * specular coefficient
    }

  if ( object transparent ) {
    Ray tRay = refraction ray
    if ( not total internal reflection ) {
       Radiance rRefr;
       rRefr = Trace ( ShadeP, tRay, depth + 1 )
       r += rRefr * transmission coefficient
       }
    }

  // return aggregate radiance
  return r;
} // Shade
```

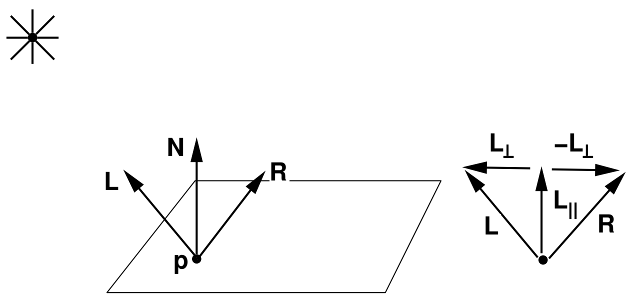

Recursive Ray Tracing: Examples

```
% ivray raycast.iv -e raycast.env -s R -d 0
% ivray raycast.iv -e raycast.env -s R -d 1
```



```
% ivray raycast.iv -e raycast.env -s R -d 2
```


Secondary Rays: Reflection

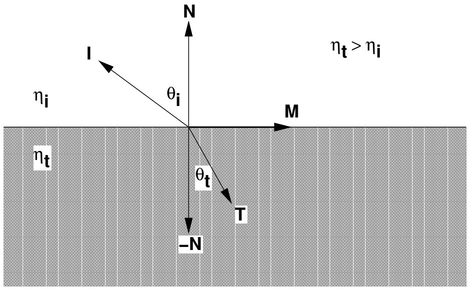


Compute reflection ray $\mathbf{R}$ as:

$$\begin{aligned}
\mathbf{L}_\parallel &= \mathbf{N}(\mathbf{L}\cdot\mathbf{N}) \\
\mathbf{R} &= \mathbf{L}_\parallel - \mathbf{L}_\perp \\
&= \mathbf{N}(\mathbf{L}\cdot\mathbf{N}) - (\mathbf{L} - \mathbf{N}(\mathbf{L}\cdot\mathbf{N})) \\
&= \mathbf{N}(\mathbf{L}\cdot\mathbf{N}) - \mathbf{L} + \mathbf{N}(\mathbf{L}\cdot\mathbf{N}) \\
&= 2\mathbf{N}(\mathbf{L}\cdot\mathbf{N}) - \mathbf{L}
\end{aligned}$$

Secondary Rays: Refraction

All media have *index of refraction* η
 ratio of c (in vacuum) / light speed (in material)
 Of course, $\eta \geq 1$ for all materials



Snell's law says that, at a boundary

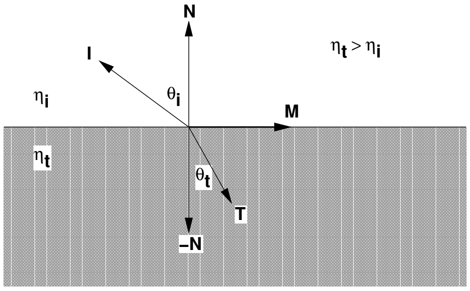
$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_t}{\eta_i}$$

or

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

 (actually, η depends on λ, causing dispersion)

Snell's law

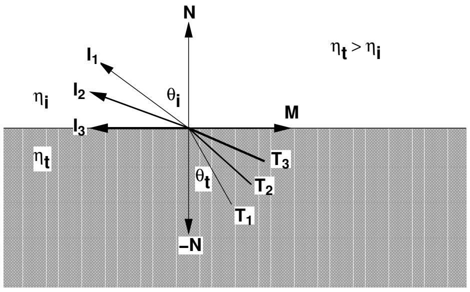


Qualitatively:
 Consider rays traveling into a denser medium
 What happens as $\eta_t \to \infty$?

$$\theta_t = \sin^{-1}\left(\frac{\eta_i}{\eta_t}\sin\theta_i\right)$$

Snell's law

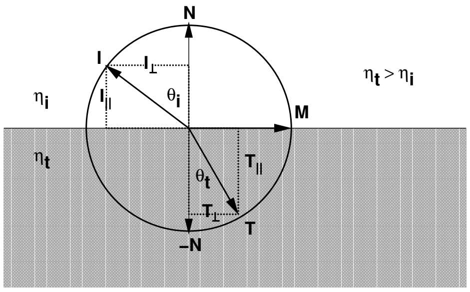
Consider rays traveling toward sparser medium



Total internal reflection at *critical angle* θ_c

$$\sin\theta_c = \left(\frac{\eta_t}{\eta_i}\right)$$

Computing Refraction Ray (Heckbert, 1990; Glassner 1994)



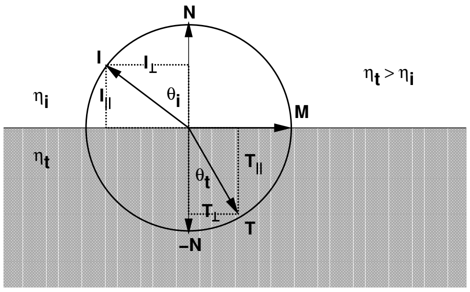
Decompose $\mathbf{I}$ into components $\mathbf{I}_\perp$ and $\mathbf{I}_\parallel$ w.r.t. $\mathbf{N}$

$$\mathbf{I}_\perp = \mathbf{N}\sin\theta_i \,; \mathbf{I}_\parallel = \mathbf{N}\cos\theta_i$$

Construct $\mathbf{M} \perp \mathbf{N}$, in plane of $\mathbf{I}$, $\mathbf{N}$:

$$\mathbf{M} = (\mathbf{N}\cos\theta_i - \mathbf{I})/\sin\theta_i$$

Refraction Ray



Decompose $\mathbf{T}$ into components $\mathbf{T}_\perp$ and $\mathbf{T}_\parallel$ w.r.t. $\mathbf{N}$
(know that $|\mathbf{T}_\perp| = \cos\theta_t$, $|\mathbf{T}_\parallel| = \sin\theta_t$)

$$\begin{aligned} \mathbf{T} &= \mathbf{T}_\perp + \mathbf{T}_\parallel \\ &= \mathbf{M}\sin\theta_t - \mathbf{N}\cos\theta_t \\ &= -\frac{\sin\theta_t}{\sin\theta_i}(\mathbf{I} - \cos\theta_i\mathbf{N}) - \mathbf{N}\cos\theta_t \end{aligned}$$

Known: $\mathbf{I}$, $\mathbf{N}$, θ_i, η_i, η_n
Solve for θ_t, plug in to find $\mathbf{T}$
But: computationally inefficient

Computing Refraction Vector

Want to express $\mathbf{T}$ in terms of $\cos\theta_i = \mathbf{N}\cdot\mathbf{I}$

Plug in $\frac{\sin\theta_t}{\sin\theta_i} = \frac{\eta_i}{\eta_t}$, collect:

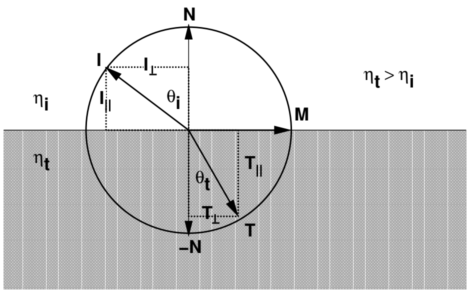
$$\begin{aligned} \mathbf{T} &= -\frac{\eta_i}{\eta_t}(\mathbf{I} - \cos\theta_i\mathbf{N}) - \mathbf{N}\cos\theta_t \\ &= -\frac{\eta_i}{\eta_t}\mathbf{I} + \mathbf{N}(\frac{\eta_i}{\eta_t}\cos\theta_i - \cos\theta_t) \end{aligned}$$

Express $\cos\theta_t$ in terms of $\cos\theta_i$:

$$\begin{aligned} \cos\theta_t &= \sqrt{1 - \sin^2\theta_t} \\ &= \sqrt{1 - (\frac{\eta_i}{\eta_t})^2 \sin^2\theta_i} \\ &= \sqrt{1 - (\frac{\eta_i}{\eta_t})^2 (1 - \cos^2\theta_i)} \end{aligned}$$

Computing Refraction Vector

Plugging it all in:



$$\mathbf{T} = -\frac{\eta_i}{\eta_t}\mathbf{I} + \mathbf{N}\left(\frac{\eta_i}{\eta_t}\cos\theta_i - \sqrt{1 - (\frac{\eta_i}{\eta_t})^2(1 - \cos^2\theta_i)}\right)$$

Eliminating cosines:

$$\mathbf{T} = -\frac{\eta_i}{\eta_t}\mathbf{I} + \mathbf{N}\left[\frac{\eta_i}{\eta_t}\mathbf{N}\cdot\mathbf{I} - \sqrt{1 - (\frac{\eta_i}{\eta_t})^2(1 - (\mathbf{N}\cdot\mathbf{I})^2)}\right]$$

What if expression under radical is negative?

Recap: Ray Tracer Components

Sample generator
One eye-ray per pixel
(You will do antialiasing in Asst 5)
Intersection finder
Workhorse function `RayCast`
Later in course, discuss acceleration techniques
Shader, Secondary Ray generator
Radiance Aggregation (base case)
Shadow (uses `RayCast` – how ?)
Reflection
Refraction

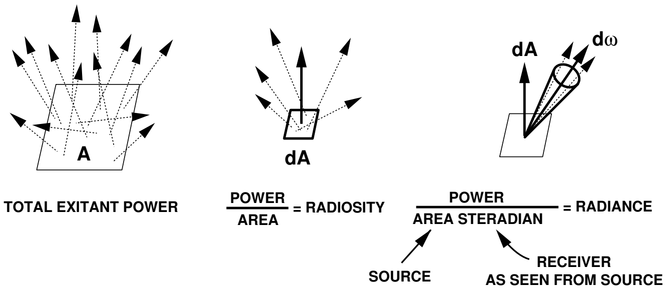
Physical Units for Ray Tracing

From radiometry, measurement of EM energy (distinct from photometry, visual sensation of EM energy)

Radiance L:

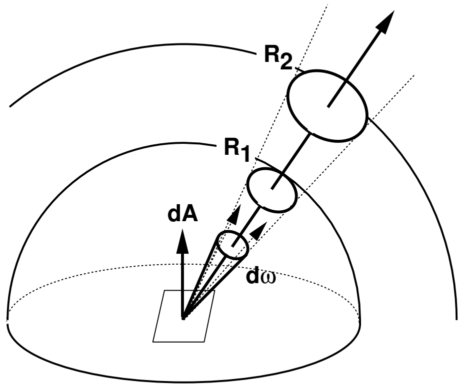
[POWER] / ([SRC AREA] [RCVR STERADIAN])

"Power per unit projected area perpendicular to the ray per unit solid angle in the direction of the ray"



Radiance Propagation

Consider two virtual spheres of radius r_1, r_2 centered at differential source element $\mathbf{dA}$, and the patches $\mathbf{R_1}, \mathbf{R_2}$ defined on them by $\mathbf{d}\omega$



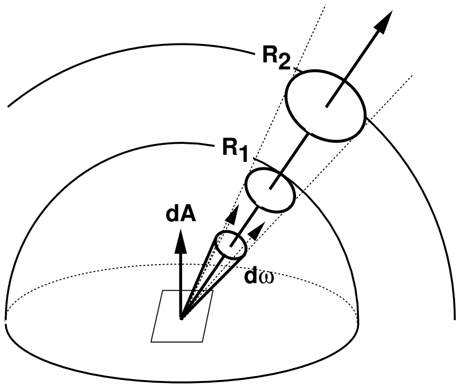
Power flowing through $\mathbf{R_1}$ is P_1, through $\mathbf{R_2}$ is P_2

$$L_1 = \frac{P_1}{A_1 \mathbf{d}\omega_\mathbf{1}} \quad L_2 = \frac{P_2}{A_2 \mathbf{d}\omega_\mathbf{2}}$$

How are L_1 and L_2 related?

Radiance Propagation

Clearly $P_1 = P_2$; $A_1 = A_2 = dA$; $\mathbf{d}\omega_\mathbf{1} = \mathbf{d}\omega_\mathbf{2}$



$$L_1 = \frac{P_1}{dA\mathbf{d}\omega_\mathbf{1}} = \frac{P_2}{dA\mathbf{d}\omega_\mathbf{2}} = L_2$$

Radiance is **constant** along a ray!

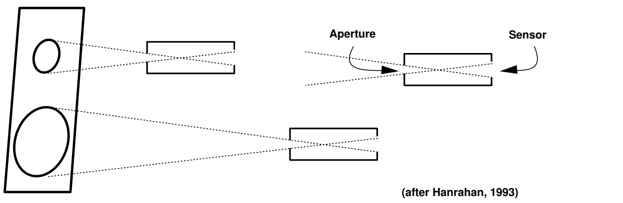
(What does this assume about propagation medium?)

Analogous derivation for fixed-size receiver

Response of a Sensor due to Radiance

Consider a small exposure meter whose field of view impinges on a large, uniformly illuminated surface

Sensors: retinal cells; film grains; CCD elements...



(after Hanrahan, 1993)

What is total POWER impinging on sensor?

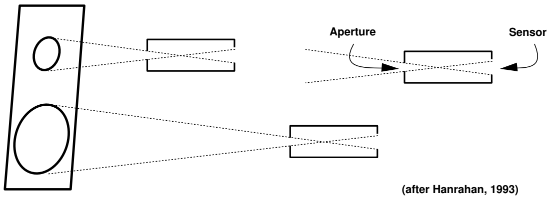
Proportional to total surface area visible to sensor

Proportional to solid angle *subtended by* sensor!

(this is just fraction of energy received by sensor!)

Response of a Sensor due to Radiance

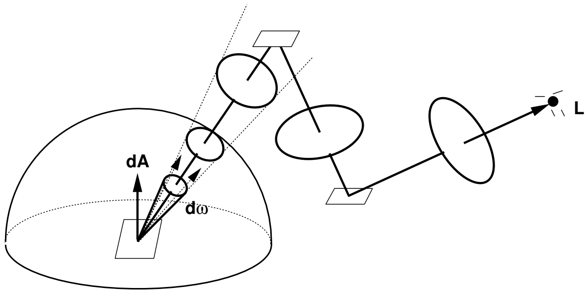
Once again, radial dependence cancels; conclude:
  Sensor response proportional to surface radiance!



Thus, for two reasons:
  RADIANCE constant along a ray
  sensor response proportional to RADIANCE
RADIANCE is the quantity that should
  be associated with a propagating ray!

Reflection, Transmission Rays

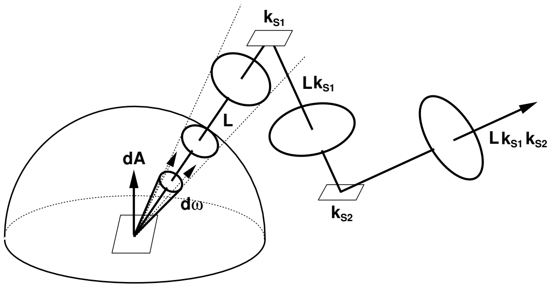
Consider a ray bouncing among *perfect mirrors*



Is there an equivalent situation with no mirrors?
Now assign mirrors coefficients of reflection k_s, k_d
  What happens?

Reflection, Transmission Rays

Radiance is *attenuated* by reflection!



Note that attenuation is **not** based on distance
  (unless, of course, the media absorbs/scatters)
Do *light*-surface distances cause attenuation?

Revised Backward RT Pseudo-Code

Want to avoid spending many cycles for little radiance
  Parameter `maxdepth` has no notion of radiance
`Trace` takes an additional parameter, `weight`
We define a new global: `float minweight`

```
global int maxdepth =  d;     // trace only d bounces, or
global float minweight = w;   // until attenuation exceeds w

RayTrace ( resolution, view frustum ) {
  For each raster y
    For each pixel x
      E = ray from eye through pixel x, y
      FrameBuffer[x][y] = Trace ( eye, E, 1, 1.0 )
} // RayTrace
```

Backward Ray Tracer Pseudo-Code

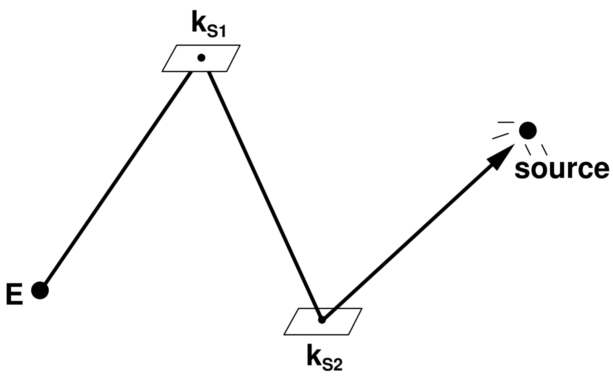
Modified definition of `Trace`:

```
Radiance Trace ( Point R, Ray D,
                 int depth, float weight ) {
  Hit *hit = RayCast ( R, D );
  if ( hit )
    return Shade ( hit->object, D, hit->P,
                   hit->N, depth, weight )
  else
    return Background ( R, D )
} // Trace
```


Revised `Shade()` Pseudo-Code

```
// Shade obj surface at point ShadeP, normal ShadeN,
//      as seen along direction Along
Radiance Shade ( Object obj, Ray Along,
                 Point ShadeP, Vector ShadeN,
                 int depth, float weight ) {
  ... // same as previous pseudo-code

  // conditionally spawn reflection ray
  if ( obj->k_s * weight >= minweight ) {
    Ray rRay = reflection ray
    radiance += obj->k_s
              * Trace ( ShadeP, rRay,
                        depth + 1, obj->k_s * weight )
    }

  // conditionally spawn transmission ray
  if ( obj->k_t * weight >= minweight ) {
    Ray tRay = refraction ray
    if ( not total internal reflection )
       radiance += obj->k_t
                 * Trace ( ShadeP, tRay,
                           depth + 1, obj->k_t * weight )
   }

  return radiance;
} // Shade
```


Call Stack




```
Trace ( R, D, 1, 1.0 );
  Shade ( object, D, R', N1, 1, 1.0 ) // 1st mirror
  // radiance += k_s1 * Trace ( R', D', 2, k_s1 )
  tmp = Trace ( R', D', 2, k_s1 )
    Shade ( object, D', R'', N2, 2, k_s1 ) // 2nd mirror
    radiance += k_s2 * f ( D', R'', N2, src )
    ...
    return radiance L; // k_s2 * ... src
  radiance += k_s1 * L;
  return radiance; // k_s2 * k_s1 * ... src
return radiance; // k_s2 * k_s1 * ... src
```


Ray Spawning / Termination

Termination conditions:
- Ray leaves the scene
- `maxdepth` exceeded
- `minweight` arises from multiple reflections

Another kind of "termination":
- Rays that are never spawned!

When might these criteria *work poorly*?